Renaming Global Variables in C Mechanically Proved Correct

Julien Cohen
Université de Nantes, France

Julien.Cohen@univ-nantes.fr

Most integrated development environments are shipped with refactoring tools. However, their refactoring operations are often known to be unreliable. As a consequence, developers have to test their code after applying an automatic refactoring. In this article, we consider a refactoring operation (renaming of global variables in C), and we prove that its core implementation preserves the set of possible behaviors of transformed programs. That proof of correctness relies on the operational semantics of C provided by CompCert C in Coq.

1 Introduction

1.1 Refactoring tools are unreliable

Designing refactoring tools is a complex task because of the complexity of the underlying programming languages. As a result, probably all refactoring tools suffer from small bugs, that occur in rare cases, but that make those tools unreliable. Many programmers have faced situations where the refactoring tool changed their program in an unexpected way. Recent works have found tens of bugs in several refactoring tools [12, 18] by systematic testing. This situation prevents programmers to trust their tools.

Some refactoring tools, such as the Haskell Refactorer [11], are rigorously designed and developed, but they are finally not free of bugs.[1]

1.2 Proving properties of refactoring tool operations

Testing and proving are complementary approaches to software validation. Although the need for safe refactoring tools is recognized [17, 3], few efforts have been done to prove the correctness of such tools.

In many research papers on refactoring, correctness is discussed informally. Some papers give formal arguments, but they either do not cover completely the preservation of behaviors (for instance: [16]) or they do not cover a complete programming language (for instance: [10]). Also, existing mechanized proofs cover only theoretical languages (for instance: [22, 21]). There is a gap between the tools that are available for mainstream languages and the tools that are reliable.

Formalized refactoring in CompCert C. Our goal in this paper is to make a step towards a fully formalized refactoring tool for an industrial language. We choose to work on the language C because it has a mechanized formalization: CompCert C [2, 9]. We present in section 2 how we use CompCert C to build a refactoring tool. Then we focus on a refactoring operation: renaming global variables. Renaming seems to be inoffensive at first sight, but we will see that there are some pitfalls to be avoided

[1] The author reported several bugs in the Haskell Refactorer (2010, 2011).

G.W. Hamilton, A. Lisitsa, A.P. Nemytykh (Eds): VPT 2016
EPTCS 216, 2016, pp. 50–64, doi:10.4204/EPTCS.216.3

(Sec. 3.1). We show how our implementation handles several situations of shadowing (Sec. 3.2) and that it preserves the behavior of programs (Sec. 3.4.1). We also give a sufficient precondition for the operation (Sec. 3.4.2).

All the properties we give are proved in Coq. The full code with the proofs is available from the author or the project web-page.

2 Refactoring in CompCert C

To be able to prove properties on the behavior of transformed programs, we need a formal definition of the semantics of programs. We rely on the semantics of C programs formalized in Coq by CompCert C, which takes into account a subset of ISO C 99 larger than MISRA C 2004[2].

That semantics (module `Csem` in CompCert) is defined on abstract syntax trees (AST). For this reason, we focus on AST transformations in this paper. This allows to verify the logic of the transformation, but it also has some limits as described below (Sec. 2.4).

2.1 CompCert C syntax and semantics

Abstract Syntax Trees. Identifiers (`AST.ident`) are represented by integers (`BinNums.positive`). A map from textual identifiers in the original program to integer identifiers is maintained during parsing (function `intern_string` defined in the OCaml module `Camlcoq`).

Programs (`AST.program`) are represented by a list of definitions (or declarations) of global variables and functions. A definition is represented by a pair (i,d) where i is the defined identifier and d is the content of that definition. Global variable definitions are composed of a type and an initialization (`AST.globdef`). Function definitions are composed of a return type, a list of parameters, a list of local variables and a body (`Csyntax.fundef`). Function bodies are represented by statements (`Csyntax.statement`). Statement syntax trees follow a 13 cases grammar, and rely on syntax trees of expressions (`Csyntax.expr`) that follow a 22 cases grammar. In that grammar for expressions, we find the construction `Evar (x:ident) (ty:type)` to represent local or global variables or function names.

Formal semantics. The semantics for C programs given in CompCert is based on small-step transitions (relation `Csem.step`). That relation respects the non-determinism of C programs: several transitions may be allowed from a given state.

The transitive closure of `step` is used to define the relation `Behaviors.program_behaves` between programs and their possible behaviors. Behaviors are represented by the following datatype (in module `Behaviors`):

```
Inductive program_behavior: Type :=
  | Terminates: trace -> int -> program_behavior
  | Diverges: trace -> program_behavior
  | Reacts: traceinf -> program_behavior
  | Goes_wrong: trace -> program_behavior.
```

Those behaviors embed *traces* which are finite or infinite lists of observable events (`trace` and `traceinf`).

[2] We have used CompCert C version 2.4. See http://compcert.inria.fr/compcert-C.html#subset for the list of supported features.

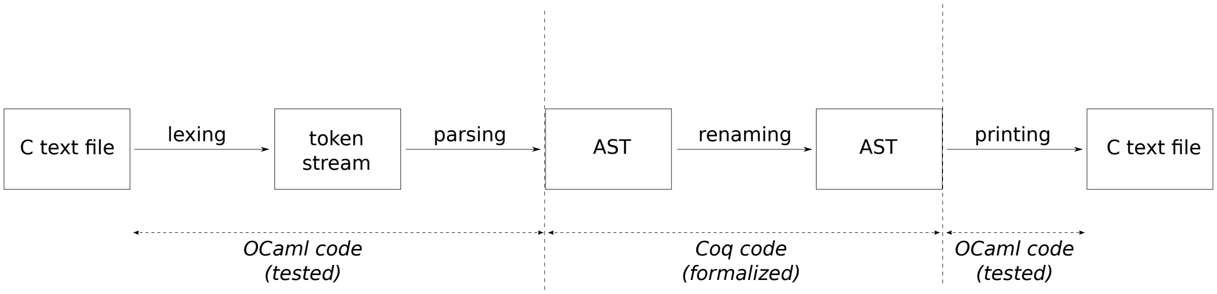

Figure 1: Data-flow of the tool.

2.2 Program Transformations

Our prototype follows the data-flow given in Fig 1. We use CompCert lexer, parser and pretty-printer. The core refactoring is performed on parsed syntax trees. Then the transformed AST are pretty-printed to recover a source file. Most refactoring tools transform simultaneously the syntax tree and the token stream in order to recover the layout in the source file, but this is out of the scope of this paper.

A program transformation may fail and return an error (`Error` constructor with a message) or succeed and return the transformed AST (`OK` constructor with the resulting program, see the module `Errors` of CompCert).

2.3 Behavior preservation

Strict preservation. Considering a transformation that successfully transforms a program `p` into a program `t_p`, the external behavior is *strictly* preserved when `p` and `t_p` have the same set of possible behaviors (type `program_behavior`) with respect to the relation `Behaviors.program_behaves` (there is as *bisimulation* between `p` and `t_p`).

Relaxed preservation. Some refactoring operations may not perform a strict behavior preservation, while still useful for users. In that case, we can precisely tell how the possible behaviors are modified. To do that, we exhibit a relation between the set of possible behaviors of `p` and the set of possible behaviors of `t_p`. For the renaming of global variables, we show in Sec. 3.1.3 that the set of behaviors is preserved up to renaming in the traces. We give a second example of relaxed preservation below.

Example. Consider the two programs below and a refactoring operation *Extract variable* that would transform the program on the left side into the one on the right side (when applied two times). The two programs do not have the same set of possible traces: the first one can print `AB` and `BA` whereas the second can only print `AB`.

```
int main(){


  return printf("A") + printf("B");
  }
```

```
int main(){
  int r1 = printf("A")   ;
  int r2 = printf("B")   ;
  return r1 + r2  ;
  }
```

Here, the provider of that refactoring operation can ensure to the user that the set of behaviors of the resulting program is *included in* (but not equal to) the set of behaviors of the original one (*backward simulation*).

Note that strict preservation and relaxed preservation deal only with external behaviors (termination, returned results, traces), as is generally accepted for refactoring operations.

2.4 Limits of the approach

Working on syntax trees implies the following limits:

- (Pre-processing.) We do not take pre-processing into account. We do not reconstruct pre-processor directives.
- (Lexing and pretty-printing.) We do not preserve layout and comments.
- (Block variables.) In CompCert syntax trees, block variables are encoded by function local variables. It makes as if all local variables of a function were declared at the beginning of its body. Since we consider only syntax trees, we have no way to consider different maskings in different blocks of functions. Because of that, our renaming operation detects some capture situations in the AST that do not occur in the text source file, and fails to perform the renaming whereas it would be legal. The following program is an example of that situation (rename *x* into *y*).

  ```
  int x = 1 ;

  void main(void){
    x++ ;
    {
      int y = 1 ;
      y++ ;
    }
  }
        /* Renaming x into y is correct */
        /* in this program.             */
  ```

  That problem does not affect the correctness of the refactoring operation but it prevents its completeness.
- Some parts of the tool (parser, pretty-printer...) cannot be proven correct in the the same framework and must be tested.
- We cannot perform renaming in programs that contain some syntax errors.

3 Renaming Global Variables

We now focus on a refactoring operation that renames global variables. Renaming is one of the refactoring operations programmers use most. Because of many shadowing and capture situations, renaming is often considered difficult.

3.1 Analysis of the problem

3.1.1 Dealing with shadowing

The interesting part of renaming variables is dealing with shadowing (a local variable *shadows* a global variable when they have the same name). To be able to preserve the behavior, we cannot create captures. In the following program, the renaming of *x* into *y* must fail:

```
int x ;
int f(int y){
  return y + x ;
}
   /* Renaming x into y is NOT correct */
   /* in this program (capture).       */
```

However, we want to be able to introduce shadowings as long as they do not produce a capture. For instance, renaming *x* into *y* in the following program introduces a new shadowing, but is correct:

```
int x ;
int f(int y){
  return y + 1 ;
}
      /* Renaming x into y is correct */
      /* in this program.             */
```

3.1.2 Volatile Variables

We do not rename volatile variables because they are designed to be shared with the outside world.

3.1.3 External Functions

The problem. In C, linked libraries have access to global variables of the program as in the following example (consider the source code of the library is not available):

```
/* The library (shipped without source code) */

extern int a ;  /* This is a declaration */
                /* (not a definition).   */

void blackbox(){ a++ ; }
```

```
/* The main program */

void blackbox() ; /* External function declaration. */

int a = 0 ;       /* Global variable definition.    */

int main(){
  blackbox() ;
  return a ;
}
```

We generally cannot or do not want to propagate the renaming in libraries. Here, renaming *a* into *b* only in the main program would change the behavior (it introduces an error).[3]

Sufficient Condition as an Hypothesis. Analysis of the compiled code of libraries is out of the scope of our prototype. So we have to assume that the renamed variable is not accessed from external code. We also assume that the new name is not used for a global variable or function in the library. Those assumptions are formalized by a predicate, `extcall_additional_properties` (see module `ExtCall` in the distributed source code). That predicate is used as a precondition for our result on behavior preservation, (hypothesis EXT1 in Fig. 2). The same assumption is made for inline assembly code that

[3] For instance, Eclipse performs the faulty renaming in the source file without warning (tested with Eclipse 4.5.1).

we do not analyze (hypothesis EXT2 in Fig. 2). Those assumptions are made only for the two names involved in the renaming.

3.2 Implementation of the transformation

We now describe our implementation. In the following, we consider x is the name to be changed, y is the new name, and $x \neq y$.

Pre-operations. Before calling the transformation defined in Coq we perform the following operations (coded in OCaml):

1. Check that y is not a C keyword. This cannot be done in the Coq part because identifiers are represented by numbers in syntax trees.
2. Trigger parsing. We use the CompCert parser, but we must not use the feature of the CompCert compiler that changes the names of the variables to have a unique name for each variable. Otherwise, we would have no way to detect shadowings in the AST.

Top-level checks and transformation. When the AST is available from parsing, the transformation defined in Coq (function `rename_globvar_hard` in module `Programs`) makes the following verifications:

1. Check that x and y are different from *main*.
2. Check that x is declared as a global variable.

Then it triggers the transformation of all definitions. For each definition, the function `rename2` in module `Definitions.Def` makes the following verifications:

1. If it defines/declares x, we check that:
   (a) the definition is not for a function but a global variable ;
   (b) x does not appear in its initialization ;
   (c) y does not appear in the initialization ;
   (d) the variable is not volatile.

   Then we change the name of the definition into y.
2. Check that it does not define/declare y .
3. In other cases, propagate the renaming in the content of the definition: function or global variable initialization.

We describe next how the renaming in functions is performed.

Renaming in functions. To propagate a renaming in a function f, we first check if f binds x and y by the means of a parameter or a local variable (see `propagate_change_ident` below[4]).

[4] The notation `do s <- E1 ; return E2` is a shortcut for `match E1 with OK s => E2 | Error e => Error e end`. See the monad `Error` defined in CompCert, which is used to represent computations that can fail. The function `dec_binds` checks if a variable is bound in a function by the means of a formal parameter or a local variable.

```
Definition propagate_change_ident (x:AST.ident) (y:AST.ident) (f:Csyntax.function) :=

      if dec_binds x f
      then
        if dec_binds y f
        then OK f
        else
          if dec_appears_statement y (fn_body f)
          then Error (msg "Replacing identifier occurring in function.")
          else OK f

      else
        if dec_binds y f
        then
          if dec_appears_statement x (fn_body f)
          then Error (msg "This renaming would introduce an undesired shadowing.")
          else OK f
        else force_body x y f.
```

```
Definition force_body (x:AST.ident) (y:AST.ident) (f:Csyntax.function) :=

  do s ← Statements.change_ident x y (fn_body f) ;
  OK (mkfunction (fn_return f) (fn_callconv f) (fn_params f) (fn_vars f) s).
```

Four different situations may occur:

- *f* does not bind *x* / *f* does not bind *y*: Rename *x* into *y* in the body of *f* (function `force_body`). This will report a failure if *y* is encountered, report a success otherwise.
- *f* binds *x* / *f* binds *y*: Do not change the body of *f* because of a shadowing (success).
- *f* does not bind *x* / *f* binds *y*: If *x* appears in the body of *f*, report a failure to avoid a capture. Else do not change the body of *f* (success).
- *f* binds *x* / *f* does not bind *y*: Do not change the body of *f* because of a shadowing, but check that *y* does not occur in the body of *f*. Otherwise, we would transform an ill-formed program into a well-formed program, as for the following program:[5]

```
int x;

int f(int x){
  return y ;    /* This instance of y is free. */
}
```

Renaming in statements and expressions. Function bodies are represented by statements. To rename a statement, we just propagate the renaming to the leaves of the syntax tree that contain occurrences of

[5] For instance, for that program, some refactoring tools (such as Eclipse and Visual Assist / Whole Tomato Software for Visual Studio) will perform the renaming of the global *x* into *y*, yielding a valid program from an invalid one, which obviously changes the semantics.

variables (constructor Evar of expressions). A failure is reported if *y* is encountered as a variable name (labels are not checked).

```
Definition change_ident_untyped (x : AST.ident) (y:AST.ident) i :=
          if AST.ident_eq x i
          then OK y
          else
            if AST.ident_eq i y
            then Error (msg ”replacing identifier already occurs”)
            else OK i .
```

Optimized syntax trees. The datatype for statements has some constructors to represent optimized forms of accesses to global variables. We report a failure when those constructors are encountered while renaming a statement, but that case never occurs since we deal with not-optimized syntax trees (those optimizations take place at compilation, not at parsing). Such cases are excluded with the hypothesis RG of our result of correctness (Fig. 2).

3.3 Trace and behavior correspondence

In CompCert, traces include some references to global variables (read and write accesses). Those references help to prove the preservation of behaviors by compilation steps of the CompCert compiler. They are not really part of the external behavior: they cannot be observed externally when the variables are not volatile and when libraries do not access them.

The presence of those references makes it impossible to preserve strictly the traces when you change some global variable names. So we want to characterize precisely those changes to be able to tell if we accept them (relaxed behavior preservation as explained in Sec 2.3).

First, we build two functions that perform renaming of global variables in finite traces (`Events.rename_in_trace`) and in infinite traces (`TraceInf.rename_traceinf`). Then we use those functions to build a function (`Behaviors.rename_globvar`) that renames global variables occurring in behaviors. That function is used to prove relaxed preservation of behaviors (Sec 3.4.1, Fig 2).

3.4 Properties

3.4.1 Behavior preservation

Under the conditions already discussed (hypotheses *EXT1*, *EXT2* and *RG* in Fig. 2), and when the renaming operations succeeds, the transformed program has the same set of possible behaviors as the original program, up to renaming in the traces. The equality of the two sets comes from a double inclusion. The first inclusion (forward simulation) is formalized by the theorem *behavior_preserved_1* (Fig. 2, proved in Coq in module `Correctness`): if the original program can have a given behavior, then the transformed program can have the same behavior (up to renaming in its trace).

The second inclusion (backward simulation) is stated in the theorem *behavior_preserved_3* (Fig. 3): if the transformed program can have a behavior, then the original program can have the same behavior (up to renaming in its trace).

```
Variable x : AST.ident.
Variable y : AST.ident.

Hypothesis EXT1 :
  ∀ name sig,
    ExtCall.extcall_additional_properties (Events.external_functions_sem name sig) x y.
Hypothesis EXT2 :
  ∀ text,
    ExtCall.extcall_additional_properties (Events.inline_assembly_sem text) x y.

Variable p : program.
Hypothesis RG : RawGlobals.rawglobals p .

Theorem behavior_preserved_1 :
  x ≠ y →
  ∀ (t_p:program) ,
    Programs.rename_globvar_hard x y p = OK t_p →
    ∀ (b : program_behavior),
      program_behaves (Csem.semantics p) b →
      ∃ t_b,
        ( Behaviors.rename_globvar x y b = OK t_b ∧
          program_behaves (Csem.semantics t_p) t_b ).
```

Figure 2: Forward simulation

```
Theorem behavior_preserved_3 :
  x≠y →
  ∀ (t_p:program),
    Programs.rename_globvar_hard x y p = OK t_p →
    ∀ (t_b : program_behavior),
      program_behaves (Csem.semantics t_p) t_b →
      ∃ b,
        ( Behaviors.rename_globvar y x t_b = OK b ∧
          program_behaves (Csem.semantics p) b ).
```

Figure 3: Backward simulation (same hypoteses as for forward simulation)

Structure of the Proof. The key point to prove the result of forward simulation above is the preservation of transitions (lemma `step_commut` of Fig. 4).

To prove this, we build a correspondence between states coming from the execution of the initial program with states in the execution of the transformed program (function `change_ident` in module `State`). That correspondence relies on correspondences we build on continuations, on contexts, and on global environments. Some aspects of the proof are further discussed in App. A.

```
Lemma step_commut :
  ∀ x y ge tr_ge st1 tr_st1 tra tr_tra st2 tr_st2,

    ( ... (* same as EXT1 in Fig. 2 *) ) →
    ( ... (* same as EXT2 in Fig. 2 *) ) →

    GlobalEnv.rename_globvar x y ge = OK tr_ge →
    State.change_ident x y st1 = OK tr_st1 →
    State.change_ident x y st2 = OK tr_st2 →
    rename_in_trace x y tra = OK tr_tra →
    step ge st1 tra st2 →
    RawGlobals.rawglobals_state st1 →
    wf_state st1 →
    step tr_ge tr_st1 tr_tra tr_st2.
```

$$\begin{tikzcd}
st1 \arrow[rr, "\mathtt{step}"'] \arrow[d, "\mathtt{change\_ident}"] & tra \arrow[d, dashed, "\mathtt{rename\_in\_trace}"] & st2 \arrow[d, "\mathtt{change\_ident}"] \\
tr\_st1 \arrow[rr, "\mathtt{step}"'] & tr\_tra & tr\_st2
\end{tikzcd}$$

Figure 4: Commutativity of renaming with transitions (relation `step`)

3.4.2 Sufficient precondition (for the transformation to succeed)

The result of correctness holds when the transformation succeeds (does not fail and return an error). Here we characterize the set of programs for which it succeeds to show it does not fail without a good reason.

To help to characterize problematic situations, we introduce the predicate `covers y x f` that says that renaming `x` into `y` in a function `f` would introduce a capture.

```
Definition covers y x f :=
  binds y f ∧ ¬binds x f ∧ appears_statement x (fn_body f).
```

Note that this predicate is coherent with the two examples of program given in Sec. 3.1.1: *x* is “covered” by *y* in the first one while *x* is not “covered” by *y* in the second one.

The predicate *no_cover_in_prog* below says that no function of a program is subject to capture.

```
Definition no_cover_in_prog x y (p : program Csyntax.fundef Ctypes.type) :=
  ∀ (f : function) (i : ident),
    List.In (i, Gfun (Csyntax.Internal f)) (prog_defs p) →
    ¬Fun.covers y x f.
```

The following result *sufficient_precondition* shows which conditions are sufficient for the transformation to succeed on a given program. Some predicates that have not been explained here can be found in the source code; they have the usual meaning the reader would probably expect.

```
Theorem sufficient_precondition :
  ∀ x y p,
    x ≠ y →
    x ≠ prog_main p →
    y ≠ prog_main p →

    RawGlobals.rawglobals p →

    defines_globvar x p →
    ¬defines_globvar y p →
    ¬defines_volatile_globvar x p →

    ¬defines_func x p →
    ¬defines_func y p →

    ¬appears_free y p →
    ¬appears_free x p →

    no_cover_in_prog x y p →

    ∃ t_p, rename_globvar_hard x y p = OK t_p .
```

Of course, this precondition applies on syntax trees, so we add that the parsing has to succeed for the transformation to succeed.

3.4.3 Invertibility

The transformation is invertible in the following meaning:

```
Lemma invertibility :
  ∀ x y p r,
    rename_globvar_hard x y p = OK r →
    rename_globvar_hard y x r = OK p.
```

3.4.4 Alternate proof for backward simulation

We have two very different proofs for the backward simulation (theorems *behavior_preserved_2*, not included in the paper and *behavior_preserved_3*, Fig. 3). The first one relies on the same technique as the proof of forward simulation whereas the second one comes for free from the invertibility of the transformation and forward simulation.

4 Conclusion

4.1 Results

We have built a refactoring tool whose logic part is formally described and proved correct. Although it has some limits (layout, pre-processing directives, and separate compilation not well supported as discussed), our prototype produces C code that has the same semantics as the initial program, even when considering non-determinism.

4.2 Related work

Faced with the complexity of making proofs for large languages such as C or Java, the community of refactoring explored naturally the systematic testing of refactoring tools [12, 18] and of refactored programs [13, 7].

Authors often give some properties of their refactoring operations, but generally in an informal way. It is not surprising that the first formal works in that domain, such as [10] and [22] (the latter is mechanized with Isabelle/HOL), were applied to functional languages, where a long tradition of program transformations exist. A few authors have a formal approach with imperative programs, but they focus on specific aspects of the transformation. For instance, [16] considers as an hypothesis that refactoring operations preserve the behavior for sequential executions in order to prove the preservation of the behavior for concurrent executions. To the best of our knowledge, our work is the first to prove formally the behavior preservation for an industrial language.

4.3 Open questions and Future Work

Validation of widespread refactoring tools. Proving the correctness of refactoring operations made us point some situations that require a deep understanding of C mechanisms. For instance, the fact that any library has a direct access to all global variables of the program is unexpected by some programmers. That experience can be used to review existing refactoring tools for C.

Refactoring ill-formed programs. Some refactoring tools can perform correct transformations in ill-formed programs, for example when there is a syntax error in a part of the program that is not concerned with a local change. Ensuring behavior preservation in presence of errors is not easy because one must ensure that the part of the program which has an error is really not impacted by the change.

False-negatives. The characterization of the set of programs that make our renaming fail while it could be performed without changing the semantics, as for the problem coming from CompCert block variables as discussed in Sec. 2.4, is difficult because it cannot be done within the CompCert formalization itself.

Preserve the layout, preserve the pre-processing directives. Most refactoring tools preserve the layout of programs and our tool could probably adapted with the techniques they implement. Preserving pre-processing directives is probably more difficult. This is studied in several papers [19, 23, 6, 14, 20].

Separate compilation. A little engineering effort is required to take separate compilation into account, and in particular the use of compiled object files or libraries in projects. Standard tools like the Unix command `nm` can probably be used to check that libraries do not use the renamed variable and its new name.

We also have to take inline assembly code into account to complete the tool.

Other refactoring operations. Of course, a large set of popular refactoring operations either atomic or composed are waiting to be verified.

Some aspects of our proof rely on some characteristics of the renaming operation, such as its invertibility, or the fact that it does not change the control flow of programs. So we expect that some parts of the proofs will change for other basic (atomic) refactoring operations. However, other aspects of our proof, such as relaxed preservation, or the correspondence between execution states, can be easily reused.

A large number of interesting refactoring operations are composite: they are combinations of basic operations. One of our goal for future work is to be able to prove the correctness of composite operations and the generation of their preconditions [8, 5] and to apply it to large transformation we have described in [4] and [1].

Aknowledgements

The author is grateful to Sandrine Blazy, University of Rennes 1, for her help with the use of CompCert.

References

[1] A. Ajouli, J. Cohen & J.-C. Royer (2013): *Transformations between Composite and Visitor Implementations in Java*. In: *Software Engineering and Advanced Applications (SEAA), 2013 39th EUROMICRO Conference on*, pp. 25–32, doi:10.1109/SEAA.2013.53.

[2] Sandrine Blazy & Xavier Leroy (2009): *Mechanized semantics for the Clight subset of the C language*. *Journal of Automated Reasoning* 43(3), pp. 263–288, doi:10.1007/s10817-009-9148-3.

[3] J. Brant & F. Steimann (2015): *Refactoring Tools are Trustworthy Enough and Trust Must be Earned*. *Software, IEEE* 32(6), pp. 80–83, doi:10.1109/MS.2015.145.

[4] J. Cohen, R. Douence & A. Ajouli (2012): *Invertible Program Restructurings for Continuing Modular Maintenance*. In: *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*, pp. 347–352, doi:10.1109/CSMR.2012.42.

[5] Julien Cohen & Akram Ajouli (2013): *Practical Use of Static Composition of Refactoring Operations*. In: *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, ACM, pp. 1700–1705, doi:10.1145/2480362.2480684.

[6] A. Garrido (2005): *Program refactoring in the presence of preprocessor directives*. Ph.D. thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA. Available at `http://hdl.handle.net/2142/11082`.

[7] Xi Ge & Emerson Murphy-Hill (2014): *Manual Refactoring Changes with Automated Refactoring Validation*. In: *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, ACM, New York, NY, USA, pp. 1095–1105, doi:10.1145/2568225.2568280.

[8] Günter Kniesel & Helge Koch (2004): *Static composition of refactorings*. *Science of Computer Programming* 52(13), pp. 9–51, doi:10.1016/j.scico.2004.03.002. Special Issue on Program Transformation.

[9] Xavier Leroy (2007–2015): CompCert C web page : `http://compcert.inria.fr/compcert-C.html`.

[10] Huiqing Li & Simon Thompson (2005): *Formalisation of Haskell Refactorings*. In Marko van Eekelen & Kevin Hammond, editors: *Trends in Functional Programming*. Available at `http://www.cs.kent.ac.uk/pubs/2005/2250`.

[11] Huiqing Li, Simon Thompson & Claus Reinke (2005): *The Haskell Refactorer, HaRe, and its API*. *Electronic Notes in Theoretical Computer Science* 141(4), pp. 29–34, doi:10.1016/j.entcs.2005.02.053. Proceedings of the Fifth Workshop on Language Descriptions, Tools, and Applications (LDTA 2005).

[12] Melina Mongiovi (2011): *Safira: A Tool for Evaluating Behavior Preservation*. In: *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '11, ACM, New York, NY, USA, pp. 213–214, doi:10.1145/2048147.2048213.

[13] Melina Mongiovi, Rohit Gheyi, Gustavo Soares, Leopoldo Teixeira & Paulo Borba (2014): *Making Refactoring Safer Through Impact Analysis*. *Sci. Comput. Program.* 93, pp. 39–64, doi:10.1016/j.scico.2013.11.001.

[14] Yoann Padioleau (2009): *Parsing C/C++ Code without Pre-processing*. In Oege de Moor & Michael I. Schwartzbach, editors: *Compiler Construction*, *Lecture Notes in Computer Science* 5501, Springer Berlin Heidelberg, pp. 109–125, doi:10.1007/978-3-642-00722-4_9.

[15] F. Pfenning & C. Elliott (1988): *Higher-order Abstract Syntax*. *SIGPLAN Not.* 23(7), pp. 199–208, doi:10.1145/960116.54010.

[16] Max Schäfer, Julian Dolby, Manu Sridharan, Emina Torlak & Frank Tip (2010): *Correct Refactoring of Concurrent Java Code*. In: *ECOOP 2010 Object-Oriented Programming*, *Lecture Notes in Computer Science* 6183, Springer Berlin Heidelberg, pp. 225–249, doi:10.1007/978-3-642-14107-2_11.

[17] Max Schäfer, Torbjörn Ekman & Oege de Moor (2008): *Challenge Proposal: Verification of Refactorings*. In: *Proceedings of the 3rd Workshop on Programming Languages Meets Program Verification*, PLPV '09, ACM, New York, NY, USA, pp. 67–72, doi:10.1145/1481848.1481859.

[18] Gustavo Soares (2012): *Automated Behavioral Testing of Refactoring Engines*. In: *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '12, ACM, New York, NY, USA, pp. 105–106, doi:10.1145/2384716.2384760.

[19] D. Spinellis (2003): *Global analysis and transformations in preprocessed languages*. *Software Engineering, IEEE Transactions on* 29(11), pp. 1019–1030, doi:10.1109/TSE.2003.1245303.

[20] Diomidis Spinellis (2010): *CScout: A refactoring browser for C*. *Science of Computer Programming* 75(4), pp. 216–231, doi:10.1016/j.scico.2009.09.003. Experimental Software and Toolkits (EST 3): A special issue of the Workshop on Academic Software Development Tools and Techniques (WASDeTT 2008).

[21] Nik Sultana & Simon Thompson (2008): *A Certified Refactoring Engine*. In: *Draft Proceedings of the Ninth Symposium on Trends in Functional Programming (TFP)*.

[22] Nik Sultana & Simon Thompson (2008): *Mechanical Verification of Refactorings*. In: *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '08, ACM, New York, NY, USA, pp. 51–60, doi:10.1145/1328408.1328417.

[23] Marian Vittek (2003): *Refactoring browser with preprocessor*. In: *Software Maintenance and Reengineering, 2003. Proceedings. Seventh European Conference on*, pp. 101–110, doi:10.1109/CSMR.2003.1192417.

A Non-trivial aspects of the proof

We report here two specific aspects of the proof of correctness that deserve to be discussed.

A.1 Higher Order Contexts

The concept of context is familiar in programming language semantics. It is used to specify places where computations can occur. In CompCert, contexts are represented by native Coq functions (type $expr \to expr$), adopting the higher-order abstract syntax style [15]. Higher order data-structures are generally used because they allow to reuse the mechanisms of function application of the host language instead of re-encoding it. But they also have the well-known drawback of being difficult to inspect and transform. So, the implementation of a renaming in contexts is not trivial.

To transform contexts, we first flatten them by applying them to a fresh witness variable (type *expr*), which gives a plain expression. Then we transform that expression (normal renaming in an expression), and then we build a function by embedding a substitution mechanism where the witness variable has the role of the placeholder for the formal parameter. See the details in our source code (module Contexts).

As a result, any reasoning on context transformations in the Coq development becomes unnatural whereas most proofs on plain expressions are close to the way you would do it "on paper".

A.2 Bindings in Continuations

Continuations are another familiar concept in programming language semantics. CompCert uses them to define the small-step semantics of C. Here is an extract of the datatype for continuations in CompCert:

```
Inductive cont: Type :=
  | Kstop: cont
  | Kseq: statement -> cont -> cont
  | ...
  | Kreturn: cont -> cont
  | Kcall: function -> env -> (expr -> expr) -> type -> cont -> cont.
```

All the constructors in this datatype are linear (they take at most one continuation as parameter). This makes continuations "homomorphic" with lists.

When propagating a renaming in a continuation, dealing with bindings and shadowings is more subtle than bindings in functions. Indeed, the binders are not explicit as they are in functions. The only way to bind parameters in continuations is in the functions appearing as the first parameter of the Kcall constructor. The scope of that binding is the "segment" of that continuation which begins with the given Kcall constructor and finishes with the next Kcall, or, if the opening Kcall was the last one of the continuation, at the end of the continuation (Kstop).

Moreover, the first segment of a continuation may not begin with a Kcall constructor. In this situation, the continuation does not contain enough information to determine the bindings in that segment. That information has to be found outside of that continuation: in the state construction that embeds that continuation.

It is essential to take all this into account to correctly construct the correspondence State.change_ident of Fig. 4.